The CarrierWave Architecture Pattern

September 16, 2002

Chris K. Wensel
chris@wensel.net

1.0 Intent

To provide a simple pattern for accessing sub-graphs of semantically significant objects and values from an application server.

2.0 Motivation

Current trends in n-layer distributed architectures tend toward:

- Static and optimized schema designs at the persistence layer
- Robust and maintainable business objects and logic at the business layer
- And use-case specific interaction patterns at the presentation layer

But where the business layer uses an abstract API to interact with the persistence layer (like JDBC, JDO, etc), the presentation layer tends towards coupling to use-case specific façade interfaces to by the business layer. Worse, this coupling is by design as enforced by current EJB standards.

By describing sets of business objects as sub-graphs of the business layer "whole graph", and transferring this sub-graph as individual but related value objects, a single simplified API can be presented to any client application. Significantly, this client application may or may not be directly coupled to the business semantics of the application, thus improving code re-usability.

Note the precedence set by RDBMS systems using the SQL language for client server interactions. SQL provides a means to both describe the bounds of a result set, and the valid members of the result set via the SELECT and WHERE clauses, respectively.

Equivalent benefits can be obtained by presenting client application with a simple interface that accepts a description of the sub-graph, and attributes that describe the members of the result set. Or more specifically, a description of the graph closure from a graph origin type, and what origin instances the graph closure should be applied to.

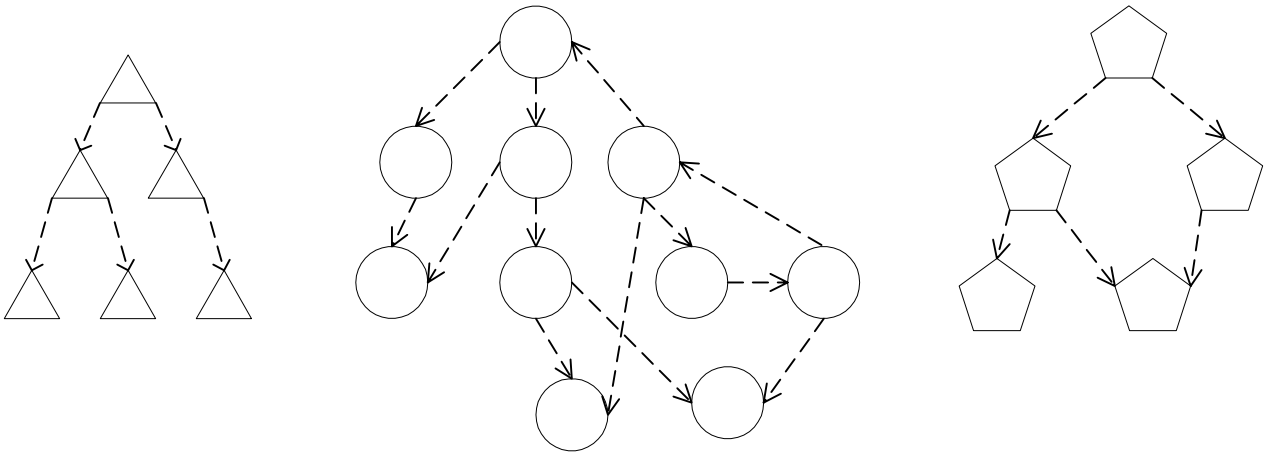
3.0 Applicability

Use this pattern when an ample number of the following statements are true:

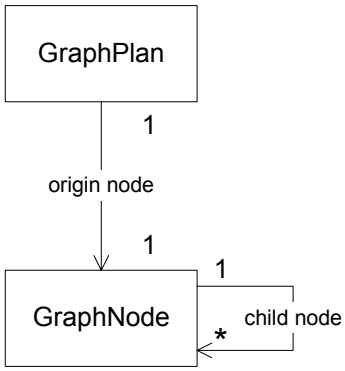
- The application uses large numbers of business objects

- Product families would benefit from sharing a “business domain agnostic” platform
- The development organizations would benefit from a hierarchical dependency between components, with shared platform development at the bottom and business domain development at the top

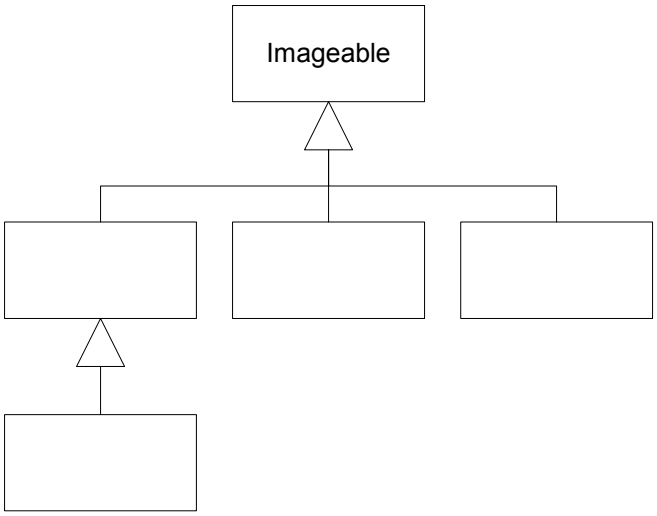
4.0 Structure



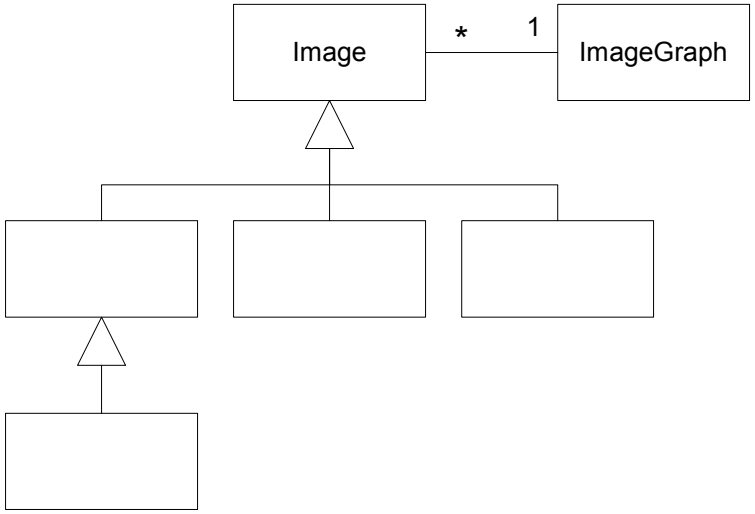
A conceptual diagram showing how graph plans are simply trees used to describe portions of whole graphs.



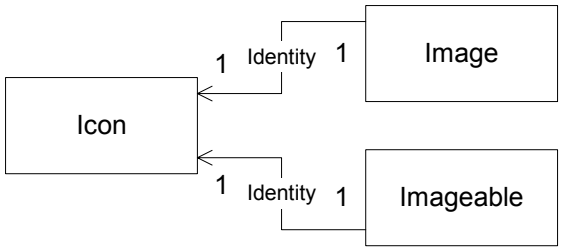
A class diagram introducing the GraphNode class GraphPlan classes. The GraphPlan is purely optional, it would allow trees of GraphNode instances to be named (versioned), persisted, and shared across components of the architecture and between threads concurrently.



A class diagram introducing the Imageable type (or possibly interface) showing that it is the root type for all business object types and sub-types.



A class diagram introducing the Image and ImageGraph types showing that the Image type is the root type for all value object types and sub-types, or more importantly, the Image type structure mirrors the Imageable type structure. The ImageGraph class manages the namespace for a give Image graph.



A class diagram introducing the Icon class and its relationship with the Image and Imageable classes. Note that two Imageable and Image instances represent the same entity if they have equivalent Icon instances.

5.0 Consequences

Key consequences of this pattern are as follows:

- Simplified remote interface: simple CRUD (create, read, update, and delete) verbs should be sufficient for most client server interactions.
- Reduced business domain coupling: developers can consciously choose to couple to business object semantics exposed by the typed Image classes, or only to the Image and ImageGraph interfaces, subsequently improving code re-usability.
- Enforced separation of concerns: by not sharing business logic with the presentation layer, presentation only code (sorting, validation, etc) and business logic code can more easily be identified and kept separate.
- Runtime efficiency: by defining the sub-graph required to render a given page or view with a graph plan, the available value objects can be retrieved in a single interaction with the application server. This is especially important when the presentation layer runs out-of-process with the application server.

6.0 Implementation

Any implementation of this pattern should consider the following:

- Default graph plans: graph plan instances can readily be generated at runtime with only the type of the origin object and a specific depth value signifying the number of graph edges to traverse.
- Reference counts: As Image instances are associated or disassociated in a given namespace (owned by an ImageGraph), child Image instances can track their number of parents. When zero, the Image instance would be an origin (has no parents) and can be made available as such via the ImageGraph interface.
- Dynamic retrieval: applying the Strategy pattern to the ImageGraph, as child Image instances are requested via getters on the parent Image, the ImageGraph can fault a locally unavailable child instance from another namespace or the remote application server.
- Few verbs: by simplifying and extending CRUD, only the following verbs should be necessary for all server interactions; select, modify, delete, and invoke.
- Dirty Images: Image instances marked dirty would improve the efficiency of an update of remote Imageable instances, if they weren't previously pruned from the source Image graph.

7.0 Related Patterns

Value Object or Transfer Object – Is a serializable class that groups related attributes, forming a composite value.